Wenhao Wu

React Native vs Flutter, cross-platform mobile application frameworks

Metropolia University of Applied Sciences

Bachelor of Engineering

Information technology

Thesis

01 March 2018

Metropolia

Author Title Number of Pages Date	Wenhao Wu React Native vs Flutter, cross-platform mobile application frameworks 28 pages 01 March 2018
Degree	Bachelor of Engineering
Degree Programme	Information technology
Professional Major	Mobile Applications
Instructors	Kari Salo, Head of Degree Programme

Developing mobile applications for Android and iOS separately has become a burden. A general solution that considers developing, maintaining, testing and deploying for different platform is an important topic. This kind of solution should be able to unify the process of mobile application development.

React Native, developed by Facebook, has been regarded as a milestone in cross-platform SDK. A strong and active community has made React Native the most popular cross-platform framework. However, after considering the pros and cons of React Native, Google has released Flutter as their approach to unite developments. Flutter is highly optimized to adapt mobile environment and aims at providing the final solution to developers.

This thesis studies some of the most important characteristics of React Native and Flutter. This research experiments and investigates the differences between those characteristics and tries to understand the reason behind them. Hopefully, React Native and Flutter will push the cross-platform framework to a new level.

Keywords	Android, iOS, React Native, Flutter

Metropolia

Contents

List of Abbreviations

OEM	Original equipment manufacturer. The original manufacturer which produce the host device.
JSX	A special JavaScript syntax extension that is used in React to describe the user interface.
DOM	Document object model. A tree-structural model where each node is an object representing a part of the document.
HTML	Hypertext markup language. A standard markup language to create the user interface of web applications.
NDK	Android native development kit. A set of tools that allows developers to program in C/C++ for Android.
LLVM	A compiler framework that complies C/C++ code into native machine code on iOS.
AOT	Ahead-of-Time compilation. An act of compilation that complies high-level programming language into native machine code.
CSS	Cascading Style Sheets. A style sheet language that is used widely in web applications.
FPS	Frame per second. A number to indicate how many frames were rendered during one second.
I/O	Input and Output. Communication between internal file system and outer data.

1 Introduction

Mobile applications are playing an increasingly important role in our daily lives. Since November 2016, there is more network traffic made by mobile devices (48.19%) compared to desktops/laptops (47%). [1]

To distribute to most of the users, a mobile application needs to adapt itself into two separate platforms, namely Android and iOS. Evidently, the differences between these two platforms are big and often require different skill sets for developing, such as Java/Kotlin only for Android and Object-C/Swift only for iOS. Thus, developers and companies often struggle at dealing with the complexity of developing cross-platform applications.

React Native, an open source cross-platform JavaScript framework, which aims at solving the above-mentioned dilemma, was introduced by Facebook on March 2015. It is based on the React framework, which is published by Facebook a few years earlier. [2] React framework is widely used by developers due to its simplicity and easy but also for its effective developing process. [3]

On the other hand, Google publishes another mobile SDK named Flutter in the end of 2016. Inspired by React Native, Flutter application can also run on both platforms, thus reducing the cost and complexity of app production across iOS and Android. Flutter is entirely built from scratch and at the time of writing this study (Aug 2017), only Google uses it for commercial project.

Cross-platform frameworks that are similar to React Native and Flutter, have been discussed and implemented by different companies several times before. However, none of them are sufficient to fulfil the industrial development requirement. Despite all those unsuccessful predecessor, React Native and Flutter, with the supports from Facebook and Google, draw a huge number of attention and people are optimistic about their prospect.

The goal of this thesis is to execute a comprehensive study on React Native and Flutter. The fundamental concepts and characteristics for both platforms will be explained and demonstrated. Comparisons, in terms of performance and developing process, between

React Native and Flutter will be covered in the thesis. Furthermore, to expose the differences between React Native and Flutter application, a fully working React Native application will be rewritten using Flutter as a supplement.

2 Theory background

2.1 React Native

This chapter introduces the history of React Native and three main characteristics of it. Also, the basic feature and structure of a React Native application will be discussed.

2.1.1 Subheading

After experiencing the huge success of React in web development, Facebook decided to expand its influence toward the mobile business. React Native started as an internal hackathon project inside of Facebook and its initial goal was to unify the development process for iOS and Android. However, as the Framework grows significantly, React Native Applicable can be deployed to other platforms, such as Windows, Web and Tizen, effortlessly.

React Native has one of the strongest communities in open-source world. In fact, currently (March 2018) React Native is the third most starred project on GitHub. Not only individual developers and Facebook are contributing to it, but also numbers of tech giants, such as Microsoft and Samsung, play important roles on developing React Native.

One of the most fascinating natures of React Native is it brings modern web techniques to mobile, without compromising much on features and performance. Even though React Native applications are mostly written in JavaScript and operate on JavaScript core, it does not mean that React Native applications are hybrid or html5 applications. The usage of underlying native interface allows React Native application to render views and access native hardware, such as camera and storage.

2.1.2 JSX

One of the easiest spot to be observed, when one studies React Native application's code, is the usage of JSX. JSX is a special syntax extension to JavaScript, which is used fundamentally to describe how the UI should show. JSX will be compiled into normal JavaScript object when the application gets compiled. A typical JSX code fragment is illustrated by figure 1.

```
action1 = () => {
  console.log("Click Button 1")
}

action2 = () => {
  console.log("Click Button 2")
}

render() {
  return (
    <View>

      <Text>
        Sample Component
      </Text>

      <TouchableOpacity onPress={this.action1}>
        <Text>
          Button1
        </Text>
      </TouchableOpacity>

      <TouchableOpacity onPress={this.action2}>
        <Text>
          Button2
        </Text>
      </TouchableOpacity>

    </View>
  )
}
```

Figure 1. Example of JSX

In figure 1, a component that contains two buttons is constructed. Two buttons handle different actions respectively. The function "render()" is crucial to every React component, since the view object of the component will be returned within. Curly brackets are used when properties of the component are required inside of the view object. Usages of arrow function in "action1()" and "action2()" are good examples of how JSX utilizes modern features of JavaScript.

Since JSX is used to describe UI, one may argue JSX is plainly another template language, such as HTML or XAML. But this is incorrect. JSX and normal JavaScript object are multi-convertible, which means we can write normal JavaScript expression in the middle of JSX.

It is worth to mention that using JSX brings the benefit of preventing injection attacks. React DOM processes any inputted value into a regular string before rendering it. Therefore, user can never inject any scripts or commands into your application by its interface [4].

2.1.3 Virtual DOM

One of the main reasons why React Native application can run on different platforms is the usage of Virtual DOM. Virtual DOM allows React to manipulate a lightweight DOM tree, that is mapped with the real DOM tree, to gain the performance boost. The workflow of Virtual DOM is described in the figure 2 below.

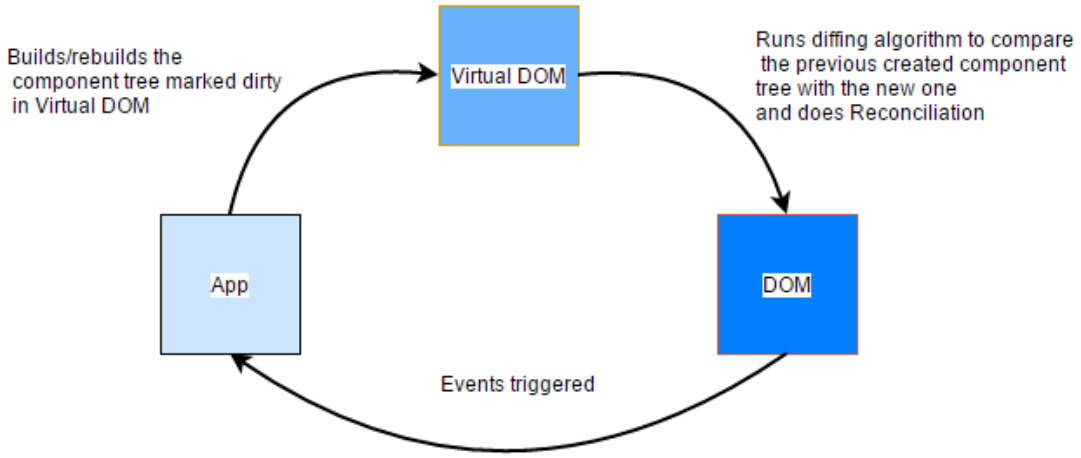

Figure 2. Workflow of Virtual DOM [5].

In figure 2, relations among application, DOM and Virtual DOM are revealed. Every user inputs to DOM, such as clicking a button, will eventually trigger events to application, which then decides the structure of Virtual DOM. The application will also run diffing algorithm periodically for updating the Real DOM effectively.

In React, whenever a JSX element needs to be rendered, its corresponding virtual DOM object will get updated at the same time. Before updating, React will take a snapshot of the current Virtual DOM tree. Therefore, after updating, React can compare the updated DOM tree with the previous snapshot to find the exact parts which need to be re-rendered in the true DOM tree. This procedure requires a set of algorithms and it is called “diffing” in the React world.

In addition to the effective “diffing” algorithm, React puts a lot of effort into batching DOM-read/write operations. After finding the minimum number of steps to update the Virtual DOM, React executes all those steps in one event loop without touching the Real DOM. And only after the event loop get finished, React will repaint the Real DOM. Thus, there should be exactly one time when the Real DOM needed to be painted [6].

Combining “diffing” and finding out which nodes in the view tree need to be updated, the whole process is called reconciliation. Naturally, in React reconciliation and rendering are two separate phases. Thus, React and React Native can use their own renders while sharing the same reconciler, provided by React core.

2.1.4 Props and State

In React, there are two major data models, which are called Props and State. They serve for different purposes and have distinct construction. Most observably, Props are set externally, and State is used only within the component’s life cycle. However, they both are plain JavaScript objects and have direct impact on triggering render updates.

Since Props is set externally by components’ parent, Props’ key function is to be used as parameters to customize the component when they are created. In other words, Props are the configuration of a component. It may be blank if it matches the needs but once it is set, it can never be changed. This nature is called immutable.

On the other hand, State could only be initialized within the component when it gets mounted. Also, one can assign a new value to component’s state whenever it suits. It is a common practice that every user’s input should have a corresponding impact to component’s State. In a complex interactive React application, component’s State often get passed as Props of its children components. Such as below:

```
<ChildComponent name={this.state.childsName} />
```

In most cases, a stateless component is easy to be tested and re-used. That is the reason why developers tend to convert parent component's state into child's component's Props.

2.2 Flutter

Similar to the previous chapter, we will discuss some of the important aspects of Flutter. The goal here is to understand the basic principle of a Flutter framework and its development cycle.

2.2.1 Introduction of Flutter

Flutter is a cross-platform framework that aims at developing high-performance mobile applications. Flutter is publicly released at 2016 by Google. Not only can Flutter applications run on Android and iOS, but also Fuschia, Google's next generation operating system, chooses Flutter as its application-level framework.

Flutter is unique. Rather than utilizing web views or relying on the device's OEM widgets, Flutter renders every view components using its own high-performance rendering engine. This nature provides possibility to build applications that are as high-performance as native applications can be. Architecture wise, the engine's C/C++ code is compiled with Android's NDK and LLVM on iOS respectively, and any Dart code is AOT-compiled into native code during compilation [13].

Flutter supports stateful hot-reload while developing, which is considered as a major factor to boost development cycle. Stateful hot-reload is essentially implemented by injecting updated source code into the running Dart VM without changing the inner structure of the application, thus all transitions and actions of the application will be preserved after hot-reloading [14].

2.2.2 Dart

In Flutter, all applications are written with Dart. Dart is a programming language that is developed and maintained by Google. It is widely used inside of Google and it has been proved to have the capability to develop massive web applications, such as AdWords.

Dart was originally developed as a replacement and successor of JavaScript. Thus, it implements most of the important characteristics of JavaScript's next standard (ES7), such as keywords "async" and "await". However, in order to attract developers that are not familiar with JavaScript, Dart has a Java-like syntax.

Akin to other systems that utilize reactive views, Flutter application refreshes the view tree on every new frame. This behaviour leads to a drawback that many objects, which may live for only one frame, will be created. Dart, as a modern programming language, is optimized to handle this scenario in memory level with the help of "Generational Garbage Collection" [7].

2.2.3 Widget

Widgets are the most important elements in a Flutter application. Widgets need to be attractive and reasonable because user see and feel them directly. Widgets do not only control and affect how the views behave, but also handle and respond to the user's action. Thus, it is crucial that Widgets need to perform fast, including rendering and animating.

Instead of reusing the OEM widgets, just as what React Native does, Flutter team decides to provide its own widgets This means Flutter, as a platform, gets to decide when and how the widgets are rendered. In a way, Flutter moves the widgets and the renderer from system level into the application itself, which allows them to be more customizable and extensible [8]. However, having the widgets and renderer within the application makes the size of application larger.

There are two major types of Widgets in Flutter, which are Stateless Widget and Stateful Widget. Table 1 below is to describe the differences between them.

	Dynamic Composition	Itself immutable	Sub State object mutable
Stateless Widget	False	True	false
Stateful Widget	True	True	True

Table 1. Comparison between widgets

Table 1 illustrates the main differences between stateful and stateless widget in Flutter. Stateful widget comes with a corresponding object that represents the state. State is considered as the presenting layer of the widget's inner structure. In plain word, state describes how widget responds to user's interactions, such as changing the widget's layout. Relatively, Stateless widget is a plain widget that does not respond to user's reaction. This design pattern allows the widget itself maintain immutable, thus preventing the framework re-renders the widget's view often.

2.3 State Management

Since React tends to be a framework which only focuses on constructing views, developers long to see helpful libraries to complete the development cycle. Redux, a library aims at helping application to maintenance its state, is beloved and recommended by the majority of the community. Due to the popularity, Flutter's official team also implements and maintains a Redux-like package for Flutter developers.

2.3.1 Widget

At the time when React was created, Facebook had a series of discussion of how an application should be structured. After investigating and considering the obstacles of developing an interactive application, unidirectional data flow, as one of the best practices, was summarized and introduced.

Traditional design pattern, such as MVC (Model-View-Controller) or MVP (Model-View-Presenter), encourages developer to decouple the business logic as an independent layer in order to increase components' reusability. However, not before long, programmers notice that soon their independent layer will hold dependencies of different views and models. Having a heavy independent layer certainly does not increase efficiency.

That's where unidirectional data flow shines. Figure 3 illustrates the structures of MVC and Flux.

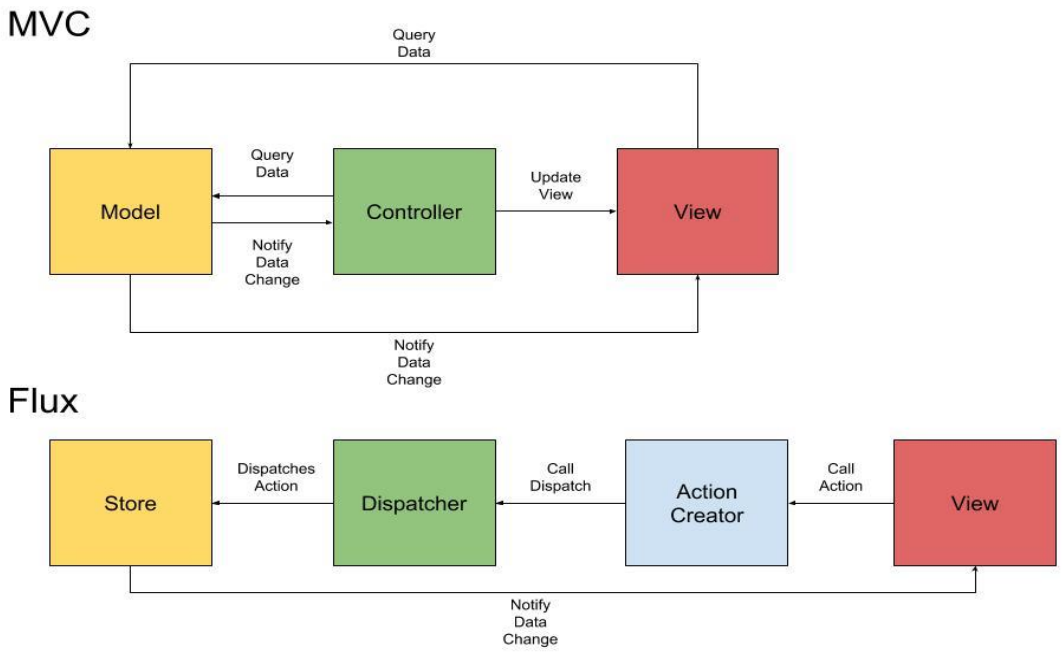

Figure 3. MVC vs Flux

By observing figure 3, distinctions between MVC and Flux are enormous. MVC allows views directly communicate with both controllers and models. This nature can break the consistency of data and raise confusion. On the contrary, Flux is an action driven architecture that honours unidirectional data flow. Each part of the architecture has and only has one input and output. Flux is implemented by Facebook aims at demonstrating how unidirectional data flow can be beneficial to applications.

2.3.2 Main priciples of Redux

One of the biggest difference between Redux and Flux is the concept of reducers. Reducers are pure functions that emit the previous state of the application with the desire action, and then output the next application state. Reducers could be described as the following equation:

$$f(state, action) = new\ state$$

On the contrary of manipulating or altering the existing state, Reducers produce a new state based on the input action. This provides two major benefits:

1. State remains immutable and become universal across all components (Single source of truth).
2. Reducer is functional and context-irrelevant, which means with the same input, there is always the same output.

These two factors make the debug process much easier and more pleasant.

Moreover, since the only way to change the state is to dispatch an action, state is predictable. It assures asynchronous events, such as network request or database IO, would never impact the states nor views directly. Also, because an action is just a pure object, it can be always stably recorded, serialized, stored or even replayed.

2.3.3 Redux with native state management

As discussed in the previous chapter, components in React Native use props and state as their own data management mechanism, and Flutter's stateful widgets delegate this process to an isolated state class. However, neither React Native nor Flutter's approach could manage state beyond one component or widget. Redux, as an application level state management, has a delicate way to inject states into components/widgets.

```
import { connect } from 'react-redux'

const ShopItem = ({item, deleteItem}) => {
    return (
        <div>
            {item.text}
            <span onClick={deleteItem}> x </span>
        </div>
    )
}

const mapStateToProps = state => {
    return {
        item : state.items[0]
    }
}

const mapDispatchToProps = dispatch => {
    return {
        deleteItem : () => dispatch({
            type : 'DELETE_ITEM'
        })
    }
}

export default connect(
    mapStateToProps,
    mapDispatchToProps
)(ShopItem)
```

Figure 4. Simplified Redux in React Native

In figure 4, a simplified redux component in React Native is declared. The component “ShopItem” is connected to the application state by calling the “connect()” function with the corresponding parameters. The function “mapStateToProps()” takes the application state as parameter and return the first item of the state’s item list. Another function “mapDispatchToProps” returns a function, which dispatch an action to delete item.

In React Native, State of the application will eventually be mapped into component’s props. We established previously that no interaction could directly affect the store. Thus, we need to have a function which can turn the current state into an object, whose keys will then be passed on as the props of the component. Furthermore, a similar function, which produces props for mapping view’s input to dispatcher, is used to dispatch action when user acts. After having these two functions inside of a component, Redux can attach the component into its cycle by utilizing the connect function.

Flutter, similarly, needs to have its widgets connected to the cycle of Redux in order to keep the state synchronized. Instead of having isolated functions for doing so, the connector component of Redux exists as a widget in Flutter. Thus, the store connectors act as wrappers, which contain the actual widget and providing either the part of the Redux's state for view building, or call-backs that trigger the action's dispatching.

2.3.4 Redux vs Mobx

Although Redux was chosen as the main design pattern for state management in our case study, there are other popular derivation of the Flux architecture that are developed and maintained by the community. Mobx is one of them.

The most noticeable difference in Mobx when comparing to Redux, is the mutability of the state. As mentioned above, instead of altering the existing state, in Redux, the reducers will generate a new state whenever receiving an action. In Mobx, components can easily be notified and directly react to state's changes. Thus, there is no need to have pure functions, such as reducers, acting as the middleware. In another word, state in Mobx is evolved when needed and components subscribe and trigger the evolvement of state.

Mobx, with the help of the built-in decorators, such as "@observable" or "@action", can be terser and tidier than Redux when developing. However, the ease and freedom of Mobx does act as a two-edged sword for developers. The lack of restriction on declaring multiple state is a plain violation of the principle "Single Source of Truth".

Moreover, since components directly observe the changes of the model, actions can be no longer a pure JavaScript object. Therefore, the possibilities of recording, serializing, storing and replaying action is gone. All above mentioned aspects raises the complexity for debugging.

In conclusion, compares to Redux, Mobx enable one to kickstart the projects with fewer code and cleaner architecture. Its flat learning curve allows itself to be easily merged into the Object-Oriented Programming environment. Nevertheless, the freedom of constructing multiple state is a significant drawback. Strongly relying on Mobx' s internal mechanism to manage state could bring chaos and difficulties when the application grows.

3 Case Study

3.1 Case Introduction

In order to demonstrate the pros and cons of developing in React Native and Flutter, an open-source React Native application is rewritten on Flutter entirely as a case study. In this section, details of implementations for both platform will be discussed and compared.

As a case study, the chosen application must be straightforward and well structured. Since React Native has a more prosperous community, such an open-source application is more accessible on React Native. Thus, the case study will be written in Flutter by referencing its React Native version.

The application contains in total four pages:

1. Home page for movie:
   This page is used as a starting point of the application. In this page we fetch data about the most popular movies, as well as the currently playing movies, from the server. User can navigate to the detail page of the movie by clicking its relevant item.

2. Home page for TV show:
   This page contains data about the most popular TV shows. It has a similar structure compares to the first page. Additionally, this page and the home page are two children of the same bottom navigation tab bar.

3. Detail page:
   This page is the page for presenting detail information of a single movie/TV show item. To construct the view, it requires an item ID to be used to fetch the data from server.

4. Search page / List page:
   The main functionality of this page is to list the search result of certain filter. It could be navigated from the main page by clicking the search button or directly the desired filter button.

3.2 Routing

Routing is one of most important specs to adjudge the structure of the application. By designing the route paths, developers could decouple the whole application into parts and clarify the logics of navigating behind it.

3.2.1 React Native Navigator

As for React Native framework, one of the core and major feature missing is a fully modernized and native experience navigation [9]. Fortunately, the community has produced and chosen a popular open-source library called “React Native Navigation (RNN)” to help developers.

In order to make RNN functional, all screen-components need to be registered as a routing-path at the application’s entry point. Since Redux was chosen as our first-class state management, Store of the app states need to be passed as a register parameter.

This is beneficial for Redux to turn navigating into dispatching an action, thus navigating could be recorded and replayed for debugging whenever needed. Below figure 5 contains the code snippet for registering component as routing paths.

```
/* eslint-disable import/prefer-default-export */
import { Navigation } from 'react-native-navigation';

import Drawer from './modules/_global/Drawer';
import Movies from './modules/movies/Movies';
import MoviesList from './modules/movies/MoviesList';
import Movie from './modules/movies/Movie';
import Search from './modules/movies/Search';

export function registerScreens(store, Provider) {
    Navigation.registerComponent('movieapp.Movie', () => Movie, store, Provider);
    Navigation.registerComponent('movieapp.Movies', () => Movies, store, Provider);
    Navigation.registerComponent('movieapp.MoviesList', () => MoviesList, store, Provider);
    Navigation.registerComponent('movieapp.Search', () => Search, store, Provider);
    Navigation.registerComponent('movieapp.Drawer', () => Drawer);
}
```

Figure 5. Registration of route path

In figure 5, all 4 screens are registered as standalone components in RNN. The “registerComponent()” function takes at least two parameter:

1. A string label, which will be used as an identifier of the component,
2. A function that returns the registered component.

Passing the store and provider of redux is necessary to emerge navigation into application's state management. As for directing within the component, methods, such as "showModal()" or "push()", that belongs to RNN are utilized. Every screen-components will receive an object called "navigator" after the registration mentioned above.

3.2.2 Flutter Navigator

Not surprisingly, In Flutter, Navigator exists as a standalone widget that controls a set of children elements with a stack discipline [10]. Full-screen widgets that user can navigate to are called "Route", and these "Route" widgets is managed by Navigator. Since Navigator functions as a stack, it is possible to push route into Navigator, and Navigator can pop route out and display it whenever needed.

```
onTap: (){
  String id = _movie.id;
  Navigator.push(context, new MaterialPageRoute(
      builder: (BuildContext context)=> new MoviePage(movieID: id)));
},
```

Figure 6. Pushing route directly into navigator widget

In figure 6, a basic action of pushing a widget to the navigator stack is demonstrated. The "push()" method requires two parameters: 1). Context of the current widget. 2). A route which indicates the next widget that is navigated to. It is common to construct the route explicitly within the push function.

Navigator, as an object, could be directly constructed by giving the current context of the widget. However, in most of the development case, using the navigator provided by the top-level container widget, such as "MaterialApp" or "WidgetApp", could be a wiser idea. One of the biggest gain of using container's navigator is the possibility to register multiple routes at the entry point of the application. Code snippet above demonstrates how we start a material app with predefined route paths.

3.3 Views

3.3.1 Modularity

Both React Native and Flutter application respect the design of modularity. Meaning instead of writing apps page by page, developers need to decouple pages into components and then assemble them according to the needs. Using the layout inspector, one can easily observe how a page is constructed by numbers of components.

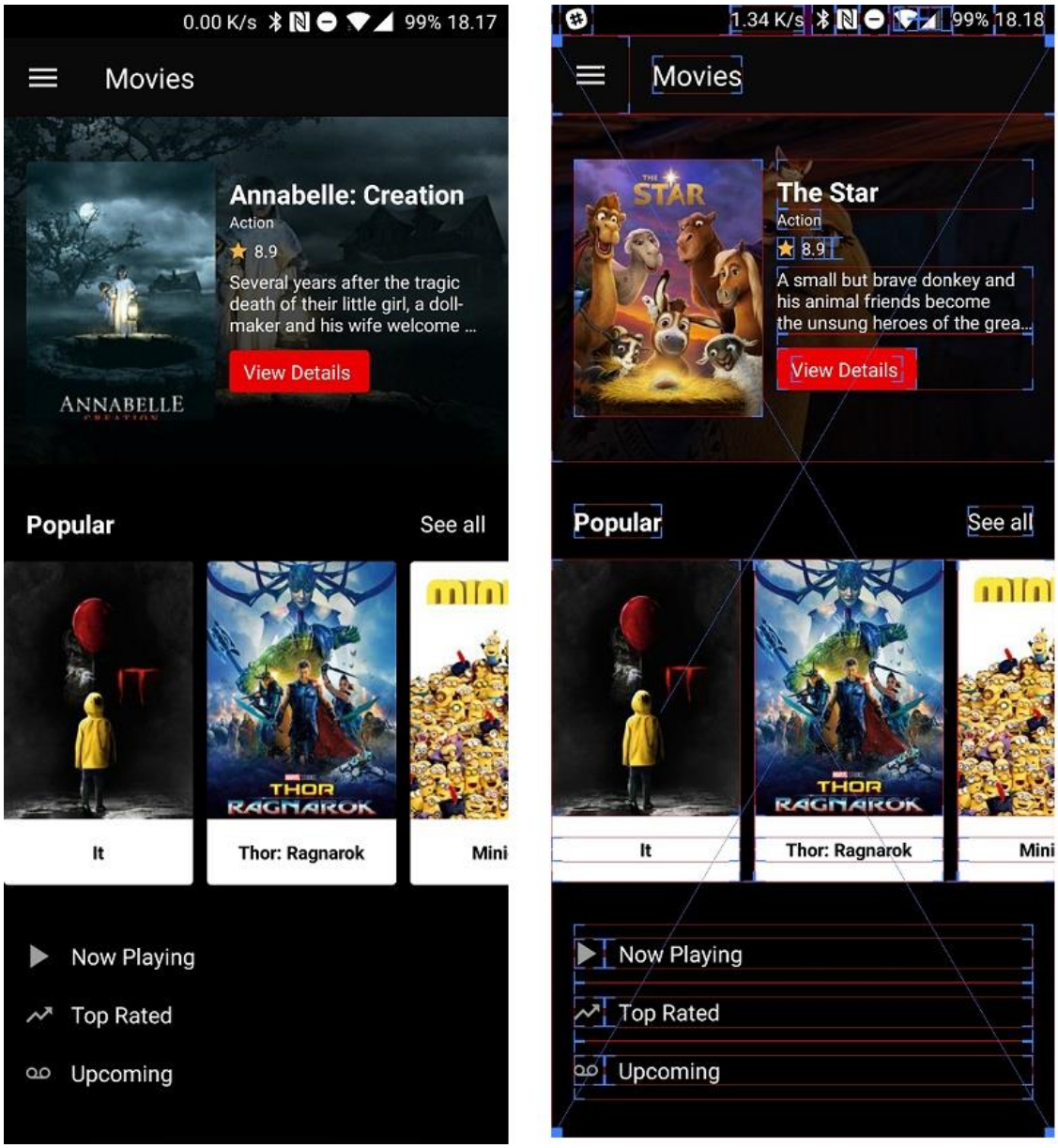

Figure 7. Layout inspector

Figure 7 indicates the usage of system layout inspector. Using inspector allows user to observe how the current page is constructed. It is observable that this example home page contains four main sections:

1. Toolbar that contains the switch of drawer.
2. Poster of the current most popular movie.
3. A horizontal list with all popular movies
4. Selections of search keywords.

It is worth mentioning that modularity also indicates reusability. A typical example is the usage of the card component in our case study. Card component is a low-level and stateless component/widget, which is self-constraint. In another word, it ensures that with the exact same input, the component will always be rendered in an identical format. To keep its nature of context-irrelevant, the styles need to be implemented and applied inside of the component.

Flutter takes the principle of modularity into another level. Not only the view component itself is a widget, every styling and rules that can be applied onto views are also widgets. Furthermore, because Flutter renders on screen directly despite of the OEM widgets, Flutter can decouple the UI toolkit from the OS. This attribute ensures the user experience and expectation are predictable and inherently stable. For example, Flutter application, which has a native iOS 11 beta look and feel, can stably function on a 7-years-old Android device.

3.3.2 Styling

React Native and Flutter holds different opinions regarding what styling essentially is and how it should be applied to views. There are historical and perspective reasons standing behind these differences. In below, we will briefly explain from which spec these distinctions come from and what problem they solve.

Not surprisingly, React, as a base framework of React Native, has a strong impact on how React Native applications are organized and developed. From the perspective of web and the influence of Cascading Style Sheet (CSS), styling is treated as primitive rules that can be applied on top of view components. However, this doesn’t mean React Native application can directly use existing CSS files. A special parser class (StyleSheet)

is created as a factory in order to generate CSS-like object, which can be furtherly applied onto view components.

On the other hand, Flutter establishes and executes resolutely the principle that widget is the one and only superclass for everything, hence stylings should be constructed and treated equally in comparison to view components. Rather than applying rules onto views, in Flutter, it is more common to build stylings and views together. A typical example could be found from the constructor of the Text widget. The constructor takes more than one parameters, which includes what to display (a string) as well as how it displays (a style object). Above code snippet reveals how it functions.

```
const myStyles = StyleSheet.create({
  red: {
    color: 'red',
  },
});

export default class StyleExample extends Component {
  render() {
    return (
      <View>
        <Text style={myStyles.red}>just red</Text>
      </View>
    );
  }
}
```

Figure 8. React Native Styling

```
const myStyles = StyleSheet.create({
  red: {
    color: 'red',
  },
});

export default class StyleExample extends Component {
  render() {
    return (
      <View>
        <Text style={myStyles.red}>just red</Text>
      </View>
    );
  }
}
```

Figure 9. Flutter Styling

Figure 8 and figure 9 together demonstrate how a red text is constructed respectively in React Native and Flutter. In React Native, style object is declared outer from the actual view component. When certain style rule is required, the view object should reference the rule from the separate style object. In Flutter, just as any other named parameter, a style object is provided for the “style” named parameter when styling is necessary.

4 Performance Comparison

4.1 Introduction

As stated in previous chapters, there are numerous differences in the implementations of Flutter and React Native. Thus, the performance of an identical scenario could vary significantly. Technically, Flutter, due to its bottom-to-top architecture, is more efficient and less resource-demanding. However, React Native has a more prosperous community. For example, mature libraries that are heavily involved in our case study, such as Redux, are developed originally for React. In all, it is unlikely to present an objective conclusion regarding performance without discussing case by case.

In order to measure the performance, some concepts need to be introduced. Frame per second (FPS), as one of the most straightforward factors to be observed, has been used widely as a standard unit to describe the fluentness of an application. One frame means one static picture of the current window. Any minor changes of the current frame will result to producing another new frame. By recording the number of rendered frames in each second on certain device, one can easily compare the performance of two different applications.

The goal of this section is to demonstrate an imperfect comparison of the performance between Flutter and React Native application. Two most-seen scenarios of mobile application are chosen as test cases for a more convincing result. All the testing and comparisons will be held on one Android device (Oneplus A3003) for monitoring the frame rate.

4.2 Scroll

Vertical Scrollable list is a typical view in all mobile applications. It is so common that both native Android and iOS provide a special set of view collections (“RecyclerView” and “UICollectionView”) to focus on optimizing the performance. Scrolling, as a simple action, requires instant feedback, animation and pre-render.

On March 2017, React Native officially deprecated “ListView” and introduced the new “FlatList” as the primary component for constructing scrollable list. The new FlatList is designed to optimize the memory usage as well as providing modern features, such as Pull to Refresh, though simplified API. Flutter, on the other hand, has not iterated its list widget often. “ListView” is built as a special child widget of the Custom “ScrollView” for displaying a set of data in linear order.

To decrease the impact of other factors, extremely simplified example is written using only framework’s component/widget. The example essentially consists a vertical scroll list with 1000 items. Each item contains an image and two lines of text. Below figure reveals some of the most important figures regarding performance.

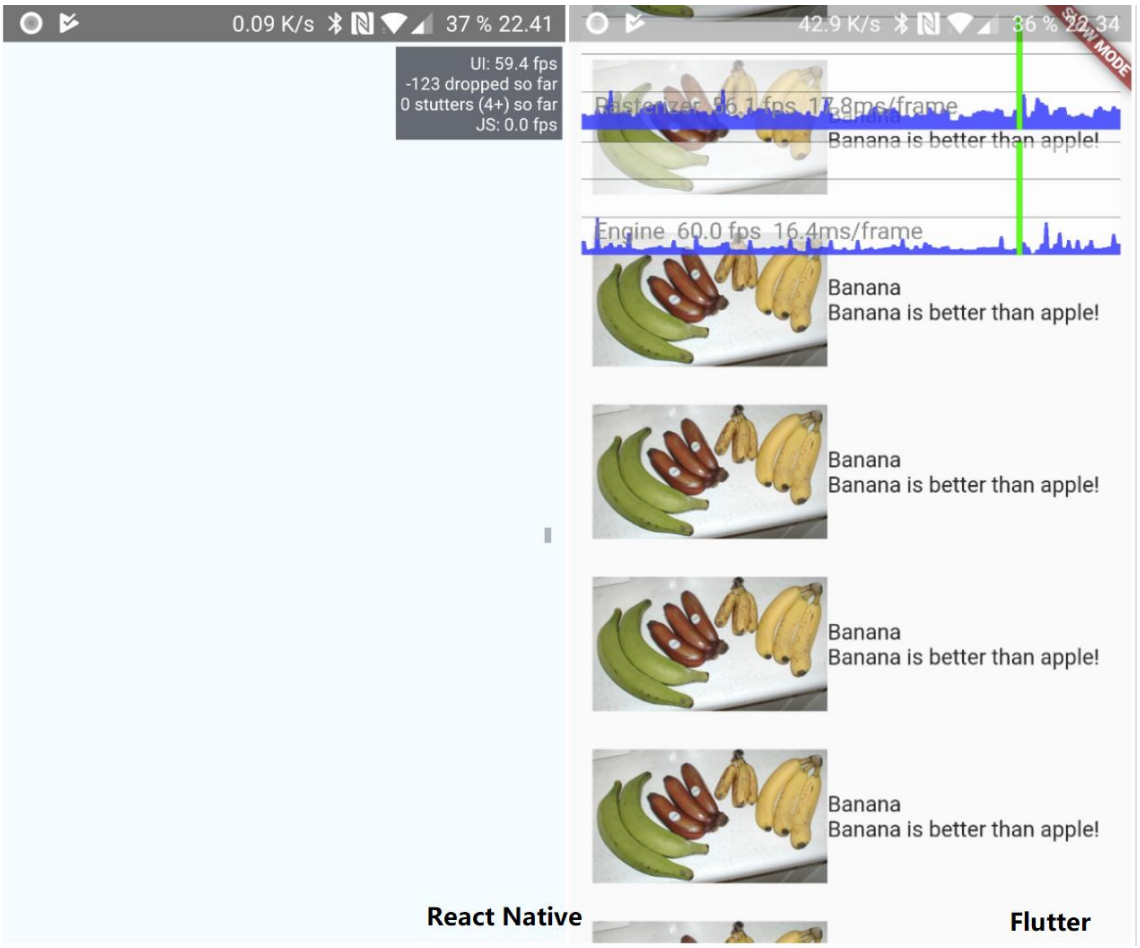

Figure 10. FPS monitor on device

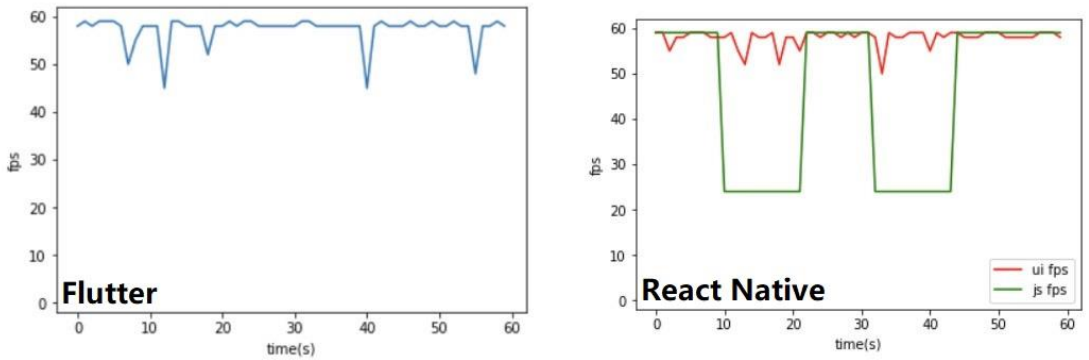

Figure 11. FPS tracking for scrolling

From both figure 10 and figure 11, one can easily notice that both Flutter and React Native do excellent job regarding to scrolling. The average fps while scrolling has been over 60 all along. However, FPS in UI thread does not fully reveal the performance for React Native. As stated in the previous chapter, majority of the React Native application runs in JavaScript thread. The bigger issue lays at how FlatList in React Native optimize

the memory usage. In order to economize memory space, for a dense list with many items, Flat list only renders certain item at once. For certain scenario, such as over speed scrolling, blank blocks will be rendered as placeholders and the fps in JavaScript thread drops significantly. On the other hand, Flutter's fps is quite stable. Through those pikes from the graph indicates handling user input and rendering animation together is resource-demanding.

4.3 Disk I/O

Another critical factor regarding to performance is the speed of input and output (I/O). File exchanges internally within the host device can be measured as the speed of disk IO. It is a common scenario for mobile application to communicate with the host device for storing data.

Most of the system file operation are handled by a library called "react-native-fs" in React Native. This library grants access to the native filesystem for React Native application. Moreover, it provides simplified API for reading and writing file asynchronously. A similar plugin can be found in Flutter's community by the name "path_provider". Both libraries utilize the native optimization of the host's device filesystem.

```
var before = new DateTime.now();
await (await _getLocalFile())
  .writeAsString('Lorem ipsum dolor sit amet')
  .whenComplete((){
    var after = new DateTime.now();
    var difference = after.difference(before);
    times.add(difference.inMilliseconds);
    int sum = 0;
    times.forEach((time){
      sum += time;
    }); // times.forEach
    var avg = sum / times.length;
    print("Average Time: $avg");
  }); // whenComplete
```

Figure 12. Core logic of measuring disk I/O speed

Figure 12 reveals the core idea of how the measurement proceed:

1. Start time counter.
2. Asynchronously opening a file whenever this function is triggered.
3. Write one sentence into the previously opened file.
4. Stop the time counter after writing went successfully.
5. Put the recorded time into the list of previously recorded time
6. Calculate the average time for each epoch.

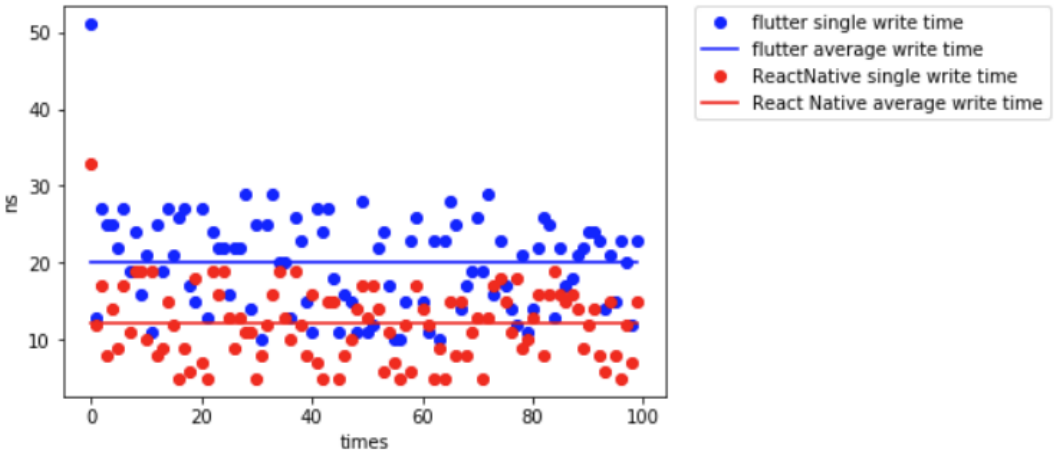

Figure 13. Char of Flutter(blue) and React Native(red) file writing consume time

As presented in the figure 13, React Native has advantage on both the average time (straight line) and single time (dots) consuming. This shall be essentially credited to the native optimization process that is done by the “react-native-fs” library. In fact, regarding to the actual writing speed, React Native ’s performance is as good as a native application can reach. Another interesting spot to observe is both Flutter and React Native use relatively long time to perform the first writing. A naive assumption would be initializing the instance of the native file I/O is a precondition to perform high speed communications.

5 Conclusion

The purpose of this thesis is to establish a comprehensive study between React Native and Flutter. During the research, theory foundation of both platforms, as well as the main characteristics, were discussed and introduced. In order to compare the process of development, a successful open-source React Native showcase is re-written in Flutter. Numerous test concerning performance of the application were carried out and analysed.

Nevertheless, due to the limitation of time and resources, not every aspect of the platforms is discussed detailly. For instance, render process, which both React Native and Flutter have put great effort into optimization, has not been deliberated. Furthermore, comparisons between native and cross-platform applications are not mentioned exhaustively.

React Native, with its pioneering work, has actively impacted the followers to certain extend. Pathbreaking concepts from React, such as one-directional dataflow and JSX, are well adopted and digested in React Native. With its strong community, React Native is no doubt the best choice to start a cross-platform application from scratch.

Flutter has a bright future. Sophisticated design from React Native are well preserved with Flutter's own evolvement. The consistency and tidiness in syntax and SDK level does bring joy to developers. Rendering widgets through a dedicate engine boosts the performance and eliminates pollutions from the OEMs.

To conclude, both React Native and Flutter have greatly proven the value of cross-platform mobile application framework. The efficiency and convenience regarding to development can surely boost the speed of pushing the product to the market. Producing a high quality and beautiful application for all mobile platforms has never been this easy before. As a trade-off, certain performance loss, when comparing to native application, is reasonably acknowledged and allowed.

References

1. Desktop vs Mobile vs Tablet Market Share Worldwide
   URL: http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet.
   Accessed July 13, 2017

2. A JavaScript Library for Building User Interfaces [Online]
   URL: https://facebook.github.io/React/.
   Accessed February 5, 2017

3. ReactJS: An Open Source JavaScript Library for Front-end Development
   URL:https://reactjs.org/
   Accessed July 13, 2017

4. Introduction JSX
   URL:https://facebook.github.io/react/docs/introducing-jsx.html
   Accessed July 15, 2017

5. How Virtual-DOM and diffing works in React
   URL:https://medium.com/@gethylgeorge/how-virtual-dom-and-diffing-works-in-react-6fc805f9f84e
   Accessed July 15, 2017

6. Virtual DOM in ReactJS
   URL:https://hackernoon.com/virtual-dom-in-reactjs-43a3fdb1d130
   Accessed July 15, 2017

7. Allocation Profile
   URL:https://dart-lang.github.io/observatory/allocation-profile.html
   Accessed September 03, 2017

8. What's Revolutionary about Flutter
   URL:https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514
   Accessed September 03, 2017

9. Official documentation of React Native Navigation
   URL:https://wix.github.io/react-native-navigation/#/
   Accessed November 03, 2017

10. Official documentation of Flutter Navigator
    URL:https://docs.flutter.io/flutter/widgets/Navigator-class.html
    Accessed November 03, 2017

11. Official React bindings for Redux

URL:https://github.com/reactjs/react-redux
Accessed December 03, 2017

12. Flutter + Redux = Fludex
    URL:https://github.com/hemanthrajv/fludex
    Accessed December 03, 2017

13. Flutter FAQ
    URL:https://github.com/hemanthrajv/fludex
    Accessed March 03, 2018

Source code

1. Flutter app for case study
   URL:https://github.com/WenhaoWu/flutter_movie
   Accessed December 03, 2017

2. React Native app for case study
   URL:https://github.com/junedomingo/movieapp
   Accessed December 03, 2017

3. Scroll FPS Comparison
   URL: https://github.com/WenhaoWu/Scroll
   Accessed February 03, 2018

4. Disk IO Comparison
   URL:https://github.com/WenhaoWu/DiskIO
   Accessed February 03, 2018